

THE PRODUCT OF ALL FEARS: BINARY MULTIPLICATION

You can perform multiplication in simple μ Ps using low-level instructions.

Modern computers typically have special instructions and employ dedicated logic (either inside or outside the CPU) to perform multiplication operations. Early microcomputers didn't have this capability, so users had limited options:

either they made do without the ability to multiply numbers (which would have restricted their ability to write useful programs to the point of absurdity, or they were obliged to come up with a technique to perform multiplication using whatever primitive instructions were available.

The shift-and-add technique

One technique for performing multiplication in any number base is by repeated addition; for example, $6 \times 4 = 6 + 6 + 6 + 6 = 24$ in decimal (similarly, $4 \times 6 = 4 + 4 + 4 + 4 + 4 + 4 = 24$). However, although computers can perform millions of operations every second, this technique can be extremely time-consuming when the values to be multiplied are large. For example, if you multiply two unsigned 16-bit numbers, in which an unsigned 16-bit number can represent values in the range 0 through 65,535 in decimal (\$0000 through \$FFFF in hexadecimal), then the worst case for a subroutine based on the repeated-addition technique would be to perform 65,535 additions, which

would be less than amusing. As an alternative, you can perform multiplications using a "shift-and-add" technique based on the generation of partial products. For example, consider the multiplication of two 8-bit unsigned binary numbers (Figure 1).

Using this algorithm, a partial product is generated for each bit in the multiplier (note that the hyphen ("-") characters in Figure 1 would actually be 0s used to pad the partial products to the necessary width). If a bit in the multiplier is 0, its corresponding partial product consists only of 0s, but if the bit in the multiplier is 1, its corresponding partial product is a copy of the multiplicand. Also, each partial product is left-shifted as a function of the multiplier bit with which the par-

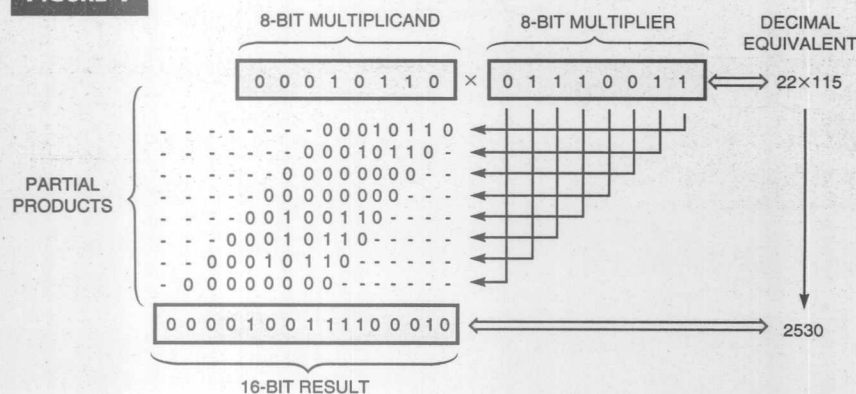
tial product is associated; for example, the partial product associated with bit 0 in the multiplier is left-shifted 0 bits; the partial product associated with bit 1 in the multiplier is left-shifted 1 bit, and so on. Once all of the partial products are generated, they are added together to generate the result. The result's width equals the sum of the widths of the multiplicand and the multiplier.

Multiplying two 8-bit numbers generates a 16-bit result (Figure 1). As each of the unsigned 8-bit numbers can carry values from 0 to 255 in decimal (\$00 through \$FF in hexadecimal), the result ranges from 0 to 65,025 in decimal (\$0000 through \$FE01 in hexadecimal). You can implement this shift-and-add approach in hardware or



CLIVE "Max" MAXFIELD
INTERGRAPH COMPUTER
SYSTEMS

FIGURE 1



This article is condensed from the forthcoming book, *Bebop Bytes Back* (Reference 1), which will be available within the next few weeks.

You can perform binary multiplication using a shift-and-add technique.

BINARY MULTIPLICATION

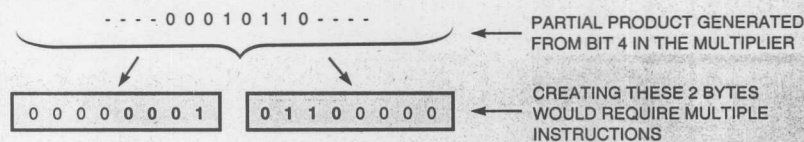
software. If you consider a hardware implementation, you can use a dedicated logic block to simultaneously generate all of the partial products and also add them. Once again, because early microcomputer users didn't have this hardware implementation, they were obliged to use a software technique, and they had to individually generate each partial product and add this product into the result. Even so, the beauty of this scheme is that it requires only the same number of additions as there are bits in the multiplier, which means eight additions for this example.

Now, there are several considerations that you need to ponder. For example, if you consider a computer with an 8-bit data bus (such as the early microcomputers), then each of the partial products in Figure 1 would be 16 bits wide (including the 0s used for padding), and you would have to perform eight 16-bit additions. Also, you would expend substantial effort splitting your multiplicand into the 8-bit quantities required to form each 16-bit partial product. Consider the partial product associated with bit 4 of the multiplier. Bits are numbered from right to left starting at 0 (Figure 2).

The problem is that you would have to perform operations to split your 8-bit multiplicand across these 2 bytes. Also, you'd have to split the multiplicand at different locations for each partial product. You can use this technique, but you might prefer a method that can achieve the same effect with fewer instructions and less hassle. In fact, there is a rather cunning ploy that you can use (Figure 3).

Initially, the two bytes representing your result are loaded with 0s. Next, you look at bit 0 in the multiplier to see if the bit contains a 0 or a 1. If bit 0 of the multiplier contains a 0, you add a byte of 0s to the most significant byte of the result, but if bit 0 of the multiplier contains a 1, then you add a copy of the multiplicand

FIGURE 2



Generating 16-bit partial products can be time-consuming if your computer has only an 8-bit data bus.

into the result's most significant byte. In both cases, you're interested in the resulting state of the carry flag in the μP 's status register.

Next, you shift both bytes of the result 1 bit to the right. When you shift the result's most significant (left) byte, you must shift the carry flag into that byte's most significant bit and also store the value of the original least significant bit that "drops off the end." Similarly, when you shift the result's least significant (right) byte, whatever bit drops off the end from the most significant byte must shift into the most significant bit of the least significant byte.

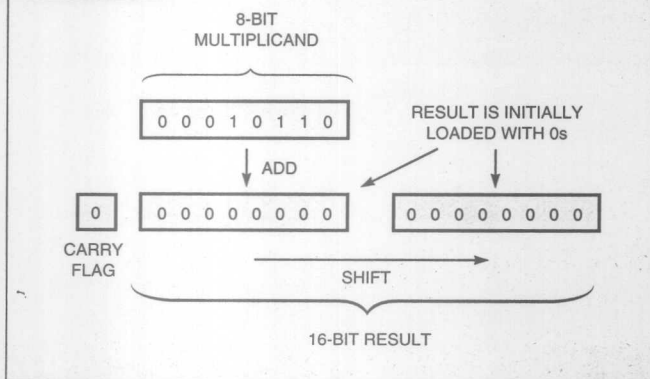
You can repeat this procedure for each of the bits in the multiplier, either adding a byte of 0s or a copy of the multiplicand into the result's most significant byte and then shifting both bytes of the result 1 bit to the right. Performing this sequence for each of the bits in the multiplier ultimately generates the

same result as does the original technique based on 16-bit partial products, but this sequence requires far fewer computer instructions.

Before you continue, there's still one more problem: extracting and testing the bits in the multiplier. One approach is to individually mask out the bits. For example, to determine the value in bit 0 of the multiplier, you can AND this value with \$01 (or %00000001 in binary) and then use a JZ ("jump if zero") instruction to vary your actions, depending on the result. Alternatively, you can use a JNZ ("jump if not zero") if this instruction better serves your purposes. Similarly, to determine the value in bit 1 of the multiplier, you can AND this value with \$02 (or %00000010 in binary); to determine the value in bit 2 of the multiplier, you can AND this value with \$04 (or %00000100 in binary); and so forth. However, you will again discover that this method requires many instructions.

As an alternative, you can simply shift the multiplier 1 bit to the right (which causes the multiplier's least significant bit to fall off the end and drop into the carry flag) and then use a JC ("jump if carry") instruction to vary your actions depending on the result. As usual, you can use a JNC ("jump if not carry") instruction if this instruction better serves your purposes. Finally, you are already shifting your 16-bit result 1 bit to the right for each bit in the multiplier, which means that whatever is initially loaded into

FIGURE 3



You can minimize your code by adding the multiplicand into the result's most significant byte and shifting to the right.

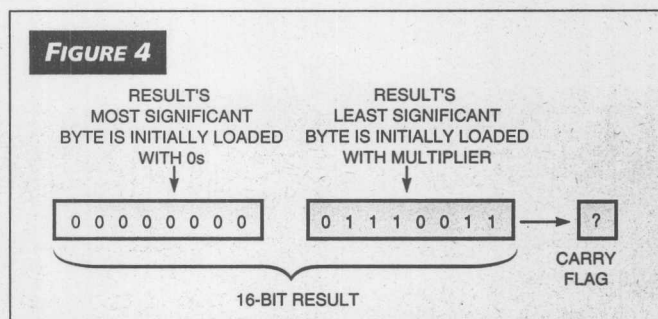
BINARY MULTIPLICATION

the result's least-significant byte ultimately will be discarded. Thus, you can save yourself effort by initializing the result so that its most significant byte contains 0s, and its least significant byte contains the multiplier (Figure 4).

Thus, every time you stroll around the main loop, which involves shifting both bytes of the result 1 bit to the right, the next bit of interest in the multiplier automatically ends up in the carry flag, ready and waiting for the following iteration of the loop. Don't worry if you find the above a bit mind-boggling at first; after a while you'll find that this "wheels-within-wheels" thinking starts to come naturally.

Unfortunately, the above technique can handle only unsigned numbers; if you consider your 8-bit numbers as representing signed values, then this routine sometimes works and sometimes doesn't, which many would consider to be less than satisfactory. Thus, if you wish to multiply signed 8-bit numbers with any confidence, you need to modify your technique.

The answer is rather simple. From the discussions above, you know how



Preloading the multiplier into the result's least significant byte also saves instructions.

to multiply unsigned (positive) numbers, so, to multiply signed (positive and negative) numbers, you must first convert any negative values into their positive equivalents and then multiply the values and correct the sign of the result if necessary. You can easily visualize the way this method works by considering a hardware implementation (Figure 5).

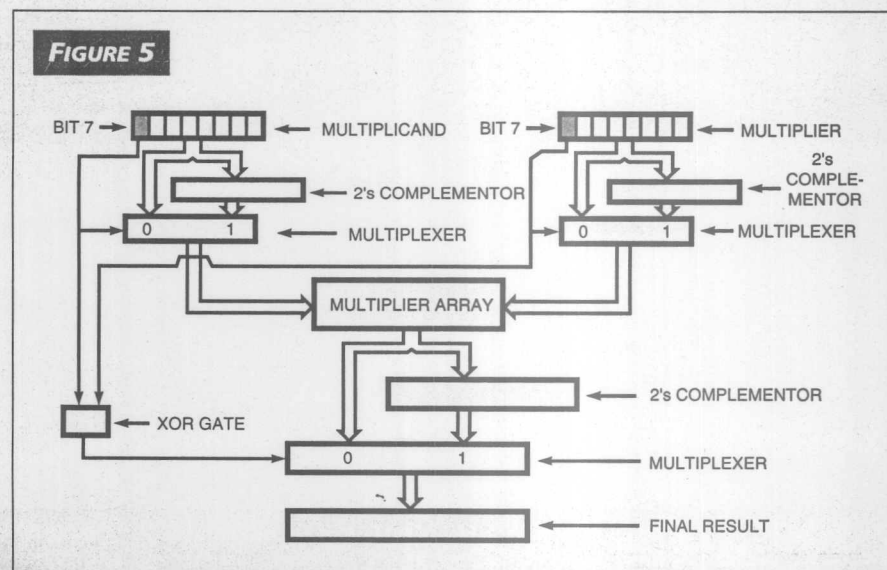
The most significant bit of a signed binary number is the *sign bit*, and this bit is *logic 0* if the value is positive and *logic 1* if the value is negative (the sign bit would be bit 7 in the case of an 8-bit number). Now, consider the multiplicand, which feeds directly into a multiplexer and into a 2's complementor. The 2's complementor automatically

generates the negative equivalent of whatever value is fed to the complementor's inputs. This means that, if the multiplicand is a positive value, the output from the 2's complementor is this value's negative equivalent, and vice versa. The output from the 2's complementor is also connected to the multiplexer, and the multiplexer's select input is driven from the multiplicand's sign bit. This

part is clever, because if the sign bit is logic 0 (indicating a positive value), this bit instructs the multiplexer to select the multiplicand. Conversely, if the sign bit is a logic 1 (indicating a negative value), this bit instructs the multiplexer to select the output from the 2's complementor, which is the positive equivalent of the multiplicand.

You can perform similar actions with the multiplier; thus, the multiplier array always receives two positive numbers, which it proceeds to multiply together. Assuming that the multiplier array is based on the shift-and-add technique, this array generates all of the partial products, adds them, and presents the output's 16-bit result.

Now comes another legerdemain, in which you decide whether to negate the output from the multiplier array. If both the multiplicand and the multiplier are positive, you need not invert the output from the array, because a positive times a positive equals a positive. Similarly, you need not invert the output from the array if both the multiplicand and the multiplier are negative, because a negative times a negative equals a positive. Thus, only when the multiplicand and the multiplier have different signs should the result be negative, in which case you need to invert the output from the array. The array's output is fed into a multiplexer and 2's complementor arrangement similar to those used for the multiplicand and the multiplier (Figure 5). However, in this case, the multiplexer's control signal comes from an XOR gate, whose inputs are the sign bits



You can easily visualize multiplying signed binary numbers as a hardware implementation.

BINARY MULTIPLICATION

from the multiplicand and the multiplier. If both sign bits have the same value, then the output from the XOR is logic 0, and the multiplexer selects the outputs from the multiplier array. Alternatively, if the sign bits have different values, then the output from the XOR is logic 1, thereby causing the multiplexer to select the outputs from the 2's complementor.

For the purposes of these discussions, you will realize a software subroutine instead of a hardware implementation, but the same general principles apply. Also, software practitioners keep a few tricks up their own sleeves; for example, to generate the 2's complement of a number you can simply subtract that number from zero; alternatively, you can achieve the same effect by inverting all of the number's bits and then adding 1. (You will see this approach in the code for the subroutine below.)

The above technique raises one slight problem. Signed 8-bit numbers can carry values from -128 through +127 in decimal (\$80 through \$7F in hexadecimal). The problem is that you cannot convert a -128 value into its positive equivalent, because your 8-bit fields cannot represent a value of +128. Thus, you must introduce a rule that says you can guarantee the results that your subroutine returns only if you present your subroutine with values in the range -127 through +127 in decimal (\$81 through \$7F in hexadecimal). It is the user's responsibility to ensure that this condition is met. Thus, the most negative result you can receive is $-127 \times +127$ (or $+127 \times -127$) = -16,129, and the most positive result is $+127 \times +127$ (or -127×-127) = +16,129, so your 16-bit result ranges from -16,129 to +16,129 in decimal (\$C0FF through \$3F01 in hexadecimal).

Next, you proceed to the assembly language (Listing 1) of the subroutine, which retrieves two 8-bit signed numbers from the stack, multiplies them, and places the 16-bit signed result on the top of the stack. (Note that this subroutine assumes that any 16-bit numbers are stored with the most significant byte "on top" of the least significant byte.)

LISTING 1—ASSEMBLY LANGUAGE OF THE SUBROUTINE

```
#####
# Copyright(c) Maxfield & Montrose Interactive Inc., 1996, 1997.
#
# The authors are not responsible for the consequences of
# using this software, no matter how awful, even if they
# arise from defects in it.
#
#####
# Name: _SMULT8
#
# Function: Multiplies two 8-bit signed numbers (in the range
#           -127 to +127) and returns a 16-bit signed result.
#
# Entry:    Top of stack
#           Most-significant byte of return address
#           Least-significant byte of return address
#           First 8-bit number (multiplicand)
#           Second 8-bit number (multiplier)
#
# Exit:     Top of stack
#           Most-significant byte of result
#           Least-significant byte of result
#
# Modifies: Accumulator
#           Index register
#
# Size:     Program = 128 bytes
#           Data = 6 bytes
#####
_SMULT8: BLDX 9                # Load the index register with 9,
                                # which equals the number of times
                                # we want to go around the loop +1

                                POPA                # Retrieve MS byte of return
                                STA [_AD_RADD]       # address from stack and store it
                                POPA                # Retrieve LS byte of return
                                STA [_AD_RADD+1]     # address from stack and store it

                                POPA                # Retrieve multiplicand from stack
                                STA [_AD_MAND]       # and store it
                                POPA                # Retrieve multiplier from stack
                                STA [_AD_RES+1]      # and store it in LS byte of result
                                LDA 0               # Load the accumulator with 0 and
                                STA [_AD_RES]        # store it in the MS byte of result

####
#### Invert input values if necessary and load the output flag
####
_AD_TSTA: LDA [_AD_MAND]       # Load the multiplicand and save
                                STA [_AD_FLAG]       # it to the flag
                                JNN [_AD_TSTB]       # If multiplicand is positive then
                                                # jump to '_AD_TSTB', otherwise ..
                                XOR $FF              # invert the contents of the ACC
                                INCA                 # ..add 1 to ACC
                                STA [_AD_MAND]       # ..store now-positive multiplicand

_AD_TSTB: LDA [_AD_FLAG]       # Load the flag,
                                XOR [_AD_RES+1]      # XOR it with the multiplier,
                                STA [_AD_FLAG]       # then store the flag again

                                LDA [_AD_RES+1]      # Load the multiplier into the ACC
                                JNN [_AD_DUMY]       # If multiplier is positive then
                                                # jump to '_AD_DUMY', otherwise ..
                                XOR $FF              # ..invert the contents of the ACC
                                INCA                 # ..add 1 to ACC
                                STA [_AD_RES+1]      # ..store now-positive multiplier

_AD_DUMY: ADD 0                # Add zero to the accumulator (dummy
                                # instruction whose sole purpose is
                                # to set the carry flag to 0)

####
#### Hold tight - this is the start of the main multiplication loop
####
```

(Listing continued on pg 202)

BINARY MULTIPLICATION

The way the flag, `_AD_FLAG`, works is as follows: In the `_AD_TSTA` part of the routine, this flag is loaded with all 8 bits of the multiplicand. Later, in the `_AD_TSTB` part of the routine, the flag, which now contains a copy of the multiplicand, is XORed with all 8 bits of the multiplier. In reality, you couldn't care less about the least significant 7 bits of the flag, because you're interested only in the most significant bit (the sign bit). However, you don't have to worry about the other bits, because when you come to actually use the flag at the beginning of the `_AD_TSTC` part of the routine, you can use a `JNN` ("jump if not negative") instruction, which considers only the state of the sign bit.

Note that this assembly language corresponds to no particular μP because the language is one that the author designed for the virtual μP implemented in software that accompanies the forthcoming book, *Bebop Bytes Back* (Reference 1). EDN

References

1. Maxfield, Clive "Max," and Alvin Brown, *Bebop Bytes Back* (An Unconventional Guide to Computers), Doone, Madison, AL.

Author's biography

Clive "Max" Maxfield, is a member of the technical staff at Intergraph Computer Systems (Huntsville, AL), (800) 763-0242, where he gets to play with the company's high-performance graphics workstations. In addition to numerous technical articles and papers, Maxfield is also the author of *Bebop* to the Boolean Boogie: An Unconventional Guide to Electronics (ISBN 1-878707-22-1). To order, phone (800) 247-6553. For more information on his forthcoming book, *Bebop Bytes Back* (An Unconventional Guide to Computers), call (800) 311-3753 or visit ro.com/~bebopbb/byte-back.htm on the Web. You can reach Maxfield via e-mail at crmaxfie@ingr.com.

VOTE

Please use the Information Retrieval Service card to rate this article (circle one):

High Interest 586	Medium Interest 587	Low Interest 588
-------------------------	---------------------------	------------------------

LISTING 1—ASSEMBLY LANGUAGE OF THE SUBROUTINE (CONTINUED)

```

_AD_LOOP: LDA [_AD_RES]      # Load ACC with MS byte of result
                                # (doesn't affect the carry flag)
                                # If carry=0, jump to start shifting
                                # otherwise add the multiplicand
                                # (which may modify the carry flag)
                                JNC [_AD_SHFT]
                                ADD [_AD_MAND]

_AD_SHFT: RORC                # Rotate the accumulator (MS byte of
                                # result) 1-bit right. This shifts
                                # the carry flag into the MS bit and
                                # also updates the carry flag with
                                # the bit that "falls off the end"
                                STA [_AD_RES]
                                # Now store the MS byte of result
                                # (doesn't affect the carry flag)

                                LDA [_AD_RES+1]
                                # Load ACC with LS byte of result
                                # (doesn't affect the carry flag)
                                RORC
                                # Rotate the LS byte of the result
                                # 1-bit right. This shifts the carry
                                # flag into the MS bit and also updates
                                # the carry flag with the multiplier
                                # bit that "falls off the end"
                                STA [_AD_RES+1]
                                # Now store the LS byte of result
                                # (doesn't affect the carry flag)

                                DECX
                                # Decrement the index register (which
                                # doesn't affect the carry flag). If
                                # the index register isn't 0 then jump
                                # back to the beginning of the loop
                                JNZ [_AD_LOOP]

#####
#### Breathe out - this is the end of the main multiplication loop
#### Now check the flag and negate the output result if necessary
#####
_AD_TSTC: LDA [_AD_FLAG]      # Load ACC with the flag
                                # If MS bit of flag is 0 then
                                # jump to '_AD_SAVE', otherwise ..
                                # ..load ACC with LS byte of result
                                LDA [_AD_RES+1]
                                XOR $FF
                                # ..invert the contents of the ACC
                                INCA
                                # ..add 1 to ACC (updates carry flag)
                                STA [_AD_RES+1]
                                # ..store negated LS byte (doesn't
                                # affect carry flag)
                                LDA [_AD_RES]
                                # ..load ACC with MS byte of result
                                # (doesn't affect carry flag)
                                XOR FF
                                # ..invert the contents of the ACC
                                # (doesn't affect carry flag)
                                ADDC $00
                                # ..propagate any carry from LS byte
                                STA [_AD_RES]
                                # ..store negated MS byte

#####
#### Save the result on the stack and then let's bug out of here
#####
_AD_SAVE: LDA [_AD_RES+1]     # Load ACC with LS byte of result
                                # and stick it on the stack
                                PUSHA
                                LDA [_AD_RES]
                                # Load ACC with MS byte of result
                                # and stick it on the stack
                                PUSHA
                                LDA [_AD_RADD+1]
                                # Load ACC with LS byte of return
                                # address from temp location and
                                # stick it back on the stack
                                PUSHA
                                LDA [_AD_RADD]
                                # Load ACC with MS byte of return
                                # address from temp location and
                                # stick it back on the stack
                                PUSHA

                                RTS
                                # That's it, exit the subroutine

_AD_RADD: .2BYTE              # Reserve 2-byte temp location for
                                # the return address
_AD_MAND: .BYTE               # Reserve 1-byte temp location for
                                # the multiplicand
_AD_RES: .2BYTE               # Reserve 2-byte temp location for
                                # the result
_AD_FLAG: .BYTE               # Reserve 1-byte to be used as a flag
                                # for negating the result (or not)

```